From VMs to Cloud-Native PostgreSQL in Kubernetes

A Case Study of Migrating a Medium-Sized Application

2024 David Pech

About Me

David Pech

wrike

Java

kubestronaut

ORACLE
Cloud Infrastructure

aws

B2B E-commerce Application

- 4 different projects with the same codebase
- Already containerized
- Legacy PHP7, Java for ETL and API endpoints
- Kafka (CSV to event-driven ETL in-progress)
- MongoDB, Redis

- App uses primary for 95% of queries
- Recalculate multiple times a day 15M prices + fluctuating stock levels
- Benefits based on customer order history
- >several 100M EUR annual turnover

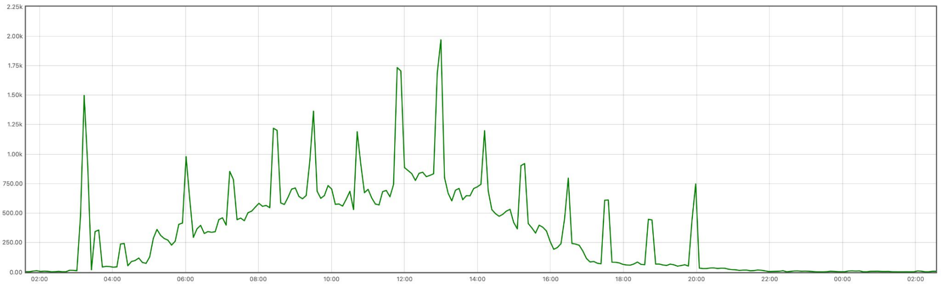
B2B E-commerce Application

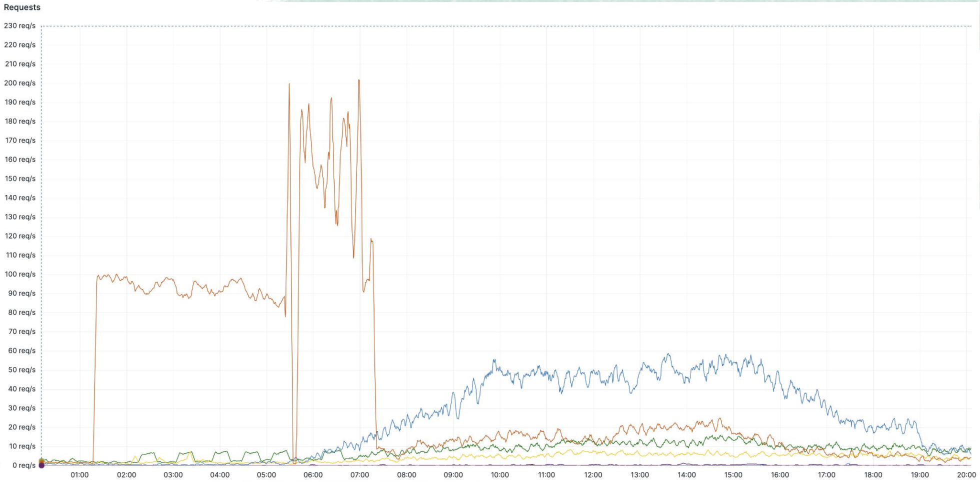
- 50 regular - 80 peak req/s for backend / project

  attacks / scans - up to 200 peak req/s, doubles before X-mas

- 2.000 regular - peak 5.000 TPS / project

  heavy caching





Organizational Context

- 0 Full-Time Postgres DBAs *(although 3 Oracle ones)*
- Application itself already migrated to K8s with success
- Client willing to invest and open to innovation
  - But running costs cheap as possible
  - (No strict SLOs)

- Unfriendly transfer of ownership from contractor
- Kubernetes adoption
- Zabbix => Prometheus migration for monitoring

Initial Postgres Setup

- OLTP 4 DBs around 70 GB each
- Traffic split: 50%, 25%, 13%, 12%
- Mixed workload of regular traffic + batch data-loading
- Ubuntu 20.04 LTS, PG13 - Version practically frozen
- no proxy / pooler
- OnPrem VmWare VMs
- Networking - directly to primary (controlled via SaltStack /etc/hosts)
- DR plan - manual, never tested on PROD
- Backups - custom pg_basebackup Bash to barman -> S3
- *Worthy mentions: pgpool-II dropped*

My Starting Point

Patroni experience:

- corrupted DB with my 1st switch-over (!)

- operating 10 DBs, internal tooling mostly
- non-trivial setup, etcd ops painful
- networking to primary

… I've never fully trusted Patroni (but probably not Patroni's problem).

Kubernetes

- operating 8 cluster OnPrem + 6 Oracle Kubernetes Engine
- Kubestronaut

Kubernetes-operators

- Bitnami chart - single instance - no-PROD
- operating 4 DBs with Zalando operator
- operating 20 DBs with CloudNative PG

Storage for K8s

- Oracle Cloud storage
- Rook/CephFS OnPrem storage

Client Motivation

- Client willing to advance technically & Good relations
- Good track record with K8s app migration (CI pipelines, ArgoCD)

- Advocating: general upgrade, H-A, logical long-term next step
- *… yet at the same time being not too critical to current setup*
- Several L1 incidents in few years, none related to Postgres (typically VmWare infra) - possible improvement with migration

=> "no big deal" from client's perspective

Client sees Kubernetes as "I can move the project to different hosting anytime".

Our Motivation

- Gradual Kubernetes adoption - stateful is next logical step
- We are not Postgres experts
- Current solution is obsolete, brings risks
- Number of services, number of users, data - grows over time

- Let's get the work done in the most reliable and stable way

~~Managed Postgres~~ vs. Patroni vs. Kubernetes-operator

Operator Research

Long story: Zalando operator, PGO, StackGres, CloudNativePG

Short story: CloudNativePG (EDB)

- Docs ++
- Enterprise-ready
- (Mature?)

DBaaS in 2024: Which PostgreSQL operator for Kubernetes to select for your platform?

David Pech

CloudNativePG (CNPG) vs. Patroni

- etcd already in K8s
- can leverage K8s nodes
- can leverage GitOps (ArgoCD)
- barman (backups)

- new tool for difficult and complex task

- basic operations can be passed to devs

- etcd operating
- need Ansible / Puppet / X node boostrap
- manual installation / first setup
- barman (backups)

- standard, **proven track record (!)**

Controlling the DB cluster

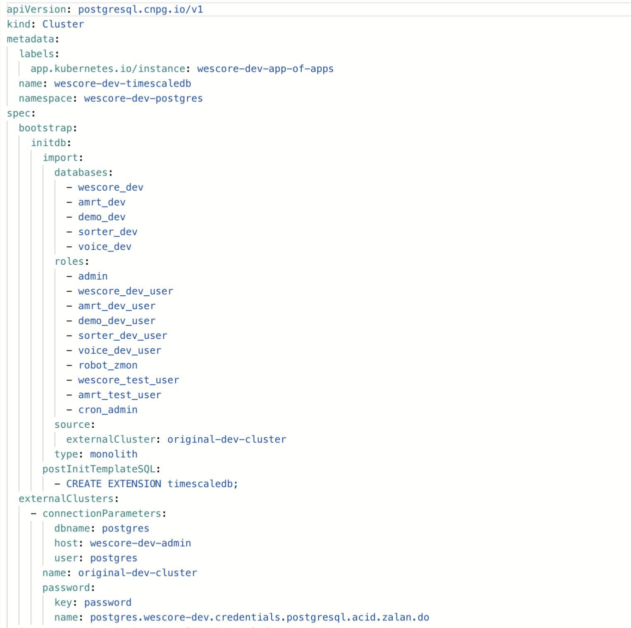
Regular operations

- operate via CustomResourceDefinition (CRD = YAML)
  - Specify users, dbs
  - Bootstrapping options
  - …
  - Change -> Edit YAML -> Operator propagates the change
- Grafana dashboard - observability

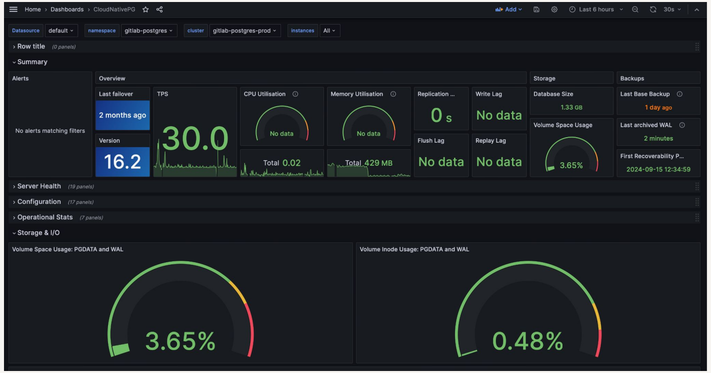
DBA

- k9s (like 'mc' for K8s)
- kubectl cnpg status
- kubectl cnpg promote

- (psql as a last option) kubectl cnpg psql (--replica)



```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  labels:
    app.kubernetes.io/instance: wescore-dev-app-of-apps
  name: wescore-dev-timescaledb
  namespace: wescore-dev-postgres
spec:
  bootstrap:
    initdb:
      import:
        databases:
          - wescore_dev
          - amrt_dev
          - demo_dev
          - sorter_dev
          - voice_dev
        roles:
          - admin
          - wescore_dev_user
          - amrt_dev_user
          - demo_dev_user
          - sorter_dev_user
          - voice_dev_user
          - robot_zmon
          - wescore_test_user
          - amrt_test_user
          - cron_admin
        source:
          externalCluster: original-dev-cluster
        type: monolith
      postInitTemplateSQL:
        - CREATE EXTENSION timescaledb;
  externalClusters:
    - connectionParameters:
        dbname: postgres
        host: wescore-dev-admin
        user: postgres
      name: original-dev-cluster
      password:
        key: password
        name: postgres.wescore-dev.credentials.postgresql.acid.zalan.do
```



Home › Dashboards › CloudNativePG
Add
Last 6 hours
30s
Row title (0 panels)
Summary
Alerts
Overview
Storage
Backups
Server Health (19 panels)
Configuration (17 panels)
Operational Stats (7 panels)
Storage & I/O

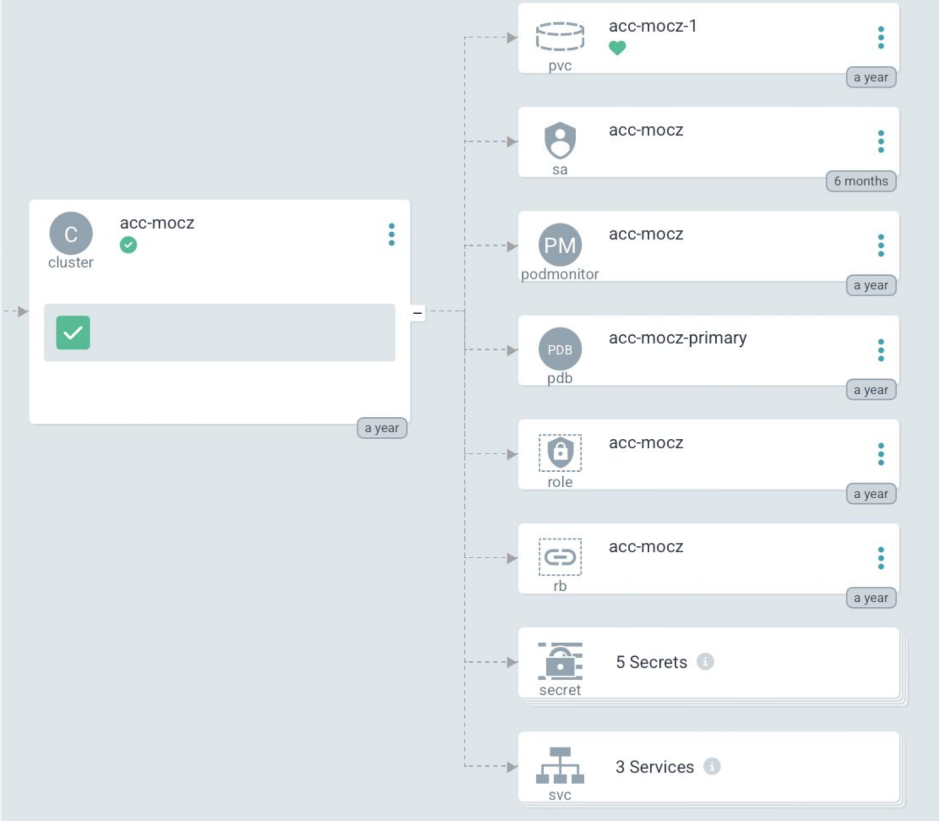
Datasource: default
namespace: gitlab-postgres
cluster: gitlab-postgres-prod
instances: All

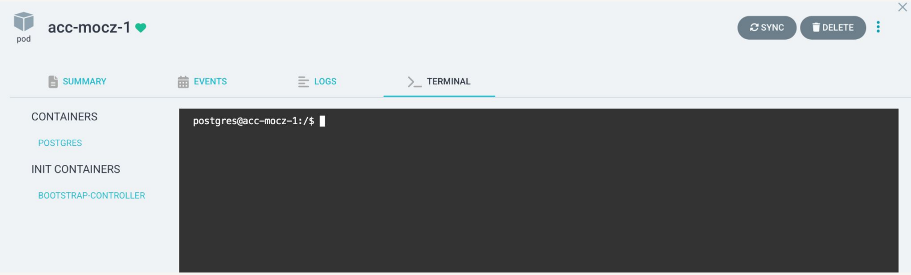
No alerts matching filters

Last failover: 2 months ago
Version: 16.2
TPS: 30.0
CPU Utilisation: No data
Total 0.02
Memory Utilisation: No data
Total 429 MB
Replication ...: 0 s
Write Lag: No data
Flush Lag: No data
Replay Lag: No data
Database Size: 1.33 GB
Volume Space Usage: 3.65%
Last Base Backup: 1 day ago
Last archived WAL: 2 minutes
First Recoverability P...: 2024-09-15 12:34:59

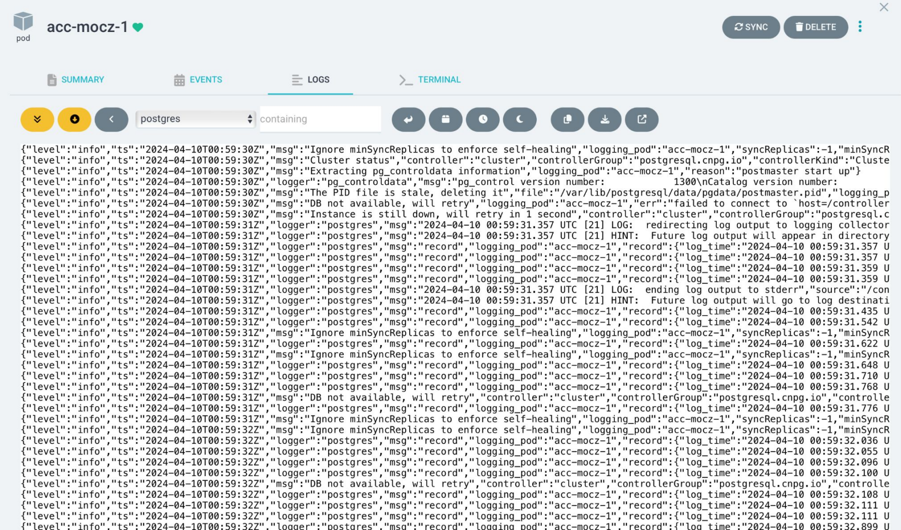
Volume Space Usage: PGDATA and WAL: 3.65%
Volume Inode Usage: PGDATA and WAL: 0.48%

Insights for Developers using ArgoCD





```
postgres@acc-mocz-1:/$
```



```
{"level":"info","ts":"2024-04-10T00:59:30Z","msg":"Ignore minSyncReplicas to enforce self-healing","logging_pod":"acc-mocz-1","syncReplicas":-1,"minSyncR
{"level":"info","ts":"2024-04-10T00:59:30Z","msg":"Cluster status","controller":"cluster","controllerGroup":"postgresql.cnpg.io","controllerKind":"Cluste
{"level":"info","ts":"2024-04-10T00:59:30Z","msg":"Extracting pg_controldata information","logging_pod":"acc-mocz-1","reason":"postmaster start up"}
{"level":"info","ts":"2024-04-10T00:59:30Z","logger":"pg_controldata","msg":"pg_control version number:            1300\nCatalog version number:
{"level":"info","ts":"2024-04-10T00:59:30Z","msg":"The PID file is stale, deleting it","file":"/var/lib/postgresql/data/pgdata/postmaster.pid","logging_p
{"level":"info","ts":"2024-04-10T00:59:30Z","msg":"DB not available, will retry","logging_pod":"acc-mocz-1","err":"failed to connect to `host=/controller
{"level":"info","ts":"2024-04-10T00:59:30Z","msg":"Instance is still down, will retry in 1 second","controller":"cluster","controllerGroup":"postgresql.c
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"2024-04-10 00:59:31.357 UTC [21] LOG:  redirecting log output to logging collector
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"2024-04-10 00:59:31.357 UTC [21] HINT:  Future log output will appear in directory
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.357 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.357 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.359 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.359 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"2024-04-10 00:59:31.357 UTC [21] LOG:  ending log output to stderr","source":"/con
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"2024-04-10 00:59:31.357 UTC [21] HINT:  Future log output will go to log destinati
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.435 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.542 U
{"level":"info","ts":"2024-04-10T00:59:31Z","msg":"Ignore minSyncReplicas to enforce self-healing","logging_pod":"acc-mocz-1","syncReplicas":-1,"minSyncR
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.622 U
{"level":"info","ts":"2024-04-10T00:59:31Z","msg":"Ignore minSyncReplicas to enforce self-healing","logging_pod":"acc-mocz-1","syncReplicas":-1,"minSyncR
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.648 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.710 U
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.768 U
{"level":"info","ts":"2024-04-10T00:59:31Z","msg":"DB not available, will retry","controller":"cluster","controllerGroup":"postgresql.cnpg.io","controlle
{"level":"info","ts":"2024-04-10T00:59:31Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:31.776 U
{"level":"info","ts":"2024-04-10T00:59:31Z","msg":"Ignore minSyncReplicas to enforce self-healing","logging_pod":"acc-mocz-1","syncReplicas":-1,"minSyncR
{"level":"info","ts":"2024-04-10T00:59:32Z","msg":"Ignore minSyncReplicas to enforce self-healing","logging_pod":"acc-mocz-1","syncReplicas":-1,"minSyncR
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.036 U
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.055 U
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.096 U
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.100 U
{"level":"info","ts":"2024-04-10T00:59:32Z","msg":"DB not available, will retry","controller":"cluster","controllerGroup":"postgresql.cnpg.io","controlle
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.108 U
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.111 U
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.111 U
{"level":"info","ts":"2024-04-10T00:59:32Z","logger":"postgres","msg":"record","logging_pod":"acc-mocz-1","record":{"log_time":"2024-04-10 00:59:32.899 U
```

Verify Operator Quality

- Reliability (Chaos) testing using Litmus + Bash

Replica recreate

Originally:

- Destroy replica VM in VmWare

Kill one the replicas together with PV (lose its data), force its reprovisioning (pg_basebackup). Wait for replica to become online before continuing.

- Metrics: Availability (Primary, Replica), Avg/Mean time to reprovision replica

Assumptions

- Primary Read-Write is not affected
- Replica is affected minimally

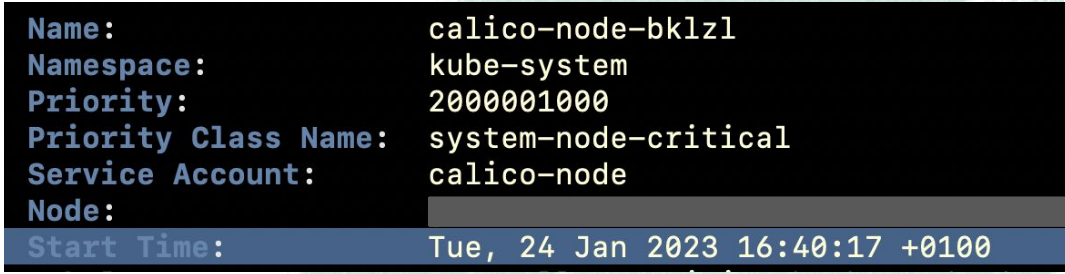
LitmusChaosCon

Will your PostgreSQL operator crack under chaos?

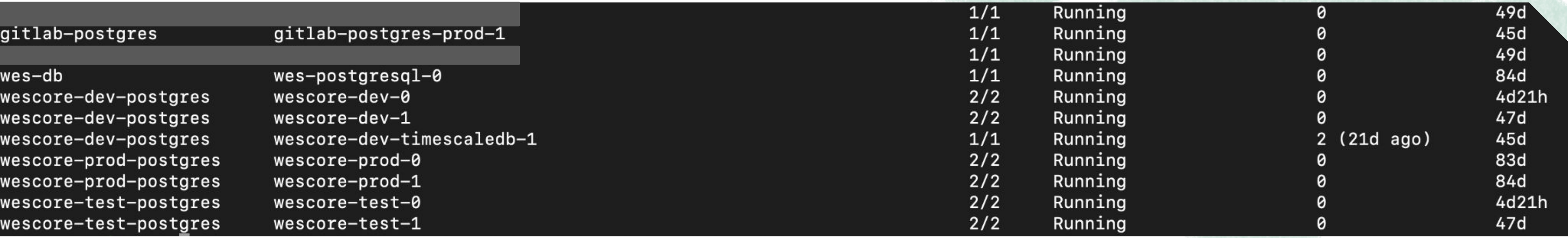
David Pech

Myth - Containers Are Ephemeral

Containers == Unix process with constraints



```
Name:                  calico-node-bklzl
Namespace:             kube-system
Priority:              2000001000
Priority Class Name:   system-node-critical
Service Account:       calico-node
Node:                  [redacted]
Start Time:            Tue, 24 Jan 2023 16:40:17 +0100
```



```
[redacted]                                                            1/1     Running        0                 49d
gitlab-postgres           gitlab-postgres-prod-1                      1/1     Running        0                 45d
[redacted]                                                            1/1     Running        0                 49d
wes-db                    wes-postgresql-0                            1/1     Running        0                 84d
wescore-dev-postgres      wescore-dev-0                               2/2     Running        0                 4d21h
wescore-dev-postgres      wescore-dev-1                               2/2     Running        0                 47d
wescore-dev-postgres      wescore-dev-timescaledb-1                   1/1     Running        2 (21d ago)       45d
wescore-prod-postgres     wescore-prod-0                              2/2     Running        0                 83d
wescore-prod-postgres     wescore-prod-1                              2/2     Running        0                 84d
wescore-test-postgres     wescore-test-0                              2/2     Running        0                 4d21h
wescore-test-postgres     wescore-test-1                              2/2     Running        0                 47d
```

Myth - Containers Are Less Performant

Prague PostgreSQL Developer Day (p2d2.cz) 2024 dialog:

"Are you considering some POC in Kubernetes?"

One of the most senior Czech PG DBAs:

"In order to run Postgres in a container, I would probably first need to 'decontantainerize it'."

Myth - Containers Are Less Performant

(Same argument was against cloud, right?)

Just untrue. Having hands-on experience needed.

Local volumes in Kubernetes = Game changer.

G. Bartolini: Local Persistent Volumes and PostgreSQL usage in Kubernetes

Myth - Kubernetes Can Easily Lose Data

Persistent Volumes (PVs) have .metadata.finalizers[]

- must be explicitly removed
- (but PVs are just YAML representation of real data somewhere)

BUT default StorageClass reclaim policy: Delete (vs. Retain)

Myth - Container Will Lose All Changes on Restart

Well, of course!

- You don't have root inside container
- Current trend: read-only root FS
- You don't use 'kubectl exec' (ssh to container)
- Container restarts with PID 1 kill

=> Design your container, Deployments etc. so they contain everything

Myth - Kubernetes Can Kill My Pod Anytime

- Well defined order of "victim selection" (preemption, PriorityClass)

- Simple rule: .resources.limits == .resources.requests

(Will make container the highest priority in "standard cluster")

- Problem:
  - .resources.requests: { cpu: 1.0, memory: 1Gi }
  - .resources.limits: { cpu: 2.0, memory: 2Gi }

(Pod might be placed to node with only 1-2Gi of free memory -> OOMKilled)

Myth - Kubernetes Needs Constant Upgrades and Breaks

- Upgrade breaking changes - significantly matured, last 2 years minimal disruptions
    - API maturity level + commitment
    - (Tooling around)

- No LTS, 3 version per year, 3 most recent version supported
    - (Yes, you ~~need to~~ should upgrade at least once a year)

Myth - Database in K8s is a Niche Idea

- Data on Kubernetes community (DoK), 2021 report

In September 2021, we surveyed over 500 Kubernetes users to understand the types and volume of data-intensive workloads being deployed in Kubernetes, benefits and challenges, and the factors driving further adoption.

…

Kubernetes has become a core part of IT – half of the respondents are running 50% or more of their production workloads on it, and they are very satisfied and more productive as a result. The most advanced users report 2x or greater productivity gains.

90% believe it is ready for stateful workloads, and a large majority (70%) are running them in production with databases topping the list. Companies report significant benefits to standardization, consistency, and management as key drivers.

Note: Nobody suggests to run 100% of your workloads in Kubernetes.

Migration Approach

- Planning
- Verify & tune solution (UAT)
- Near-zero migration on PROD
- Reliability testing

Plan: On-Prem Block Storage

- External (VmWare, Proxmox provisioner) - take it if available

- hostPath PVs
- local-path-provisioner
- Rook/Ceph - need expert know-how
    - Possibly beneficial for reads
    - Hard to setup and learn
    - Difficult to estimate or evaluate performance under load

Note: can be also static - provision PVs up-front.

Plan: On-Prem Networking

- In-cluster only, or exposed outside Kubernetes cluster?
- External HW LoadBalancers - take them if available

- kube-vip - VirtualIP
- MetalLB
- NodePort
- (not required when sharing cluster with apps)

- CNI - Cillium

Plan: Pgbouncer or not?

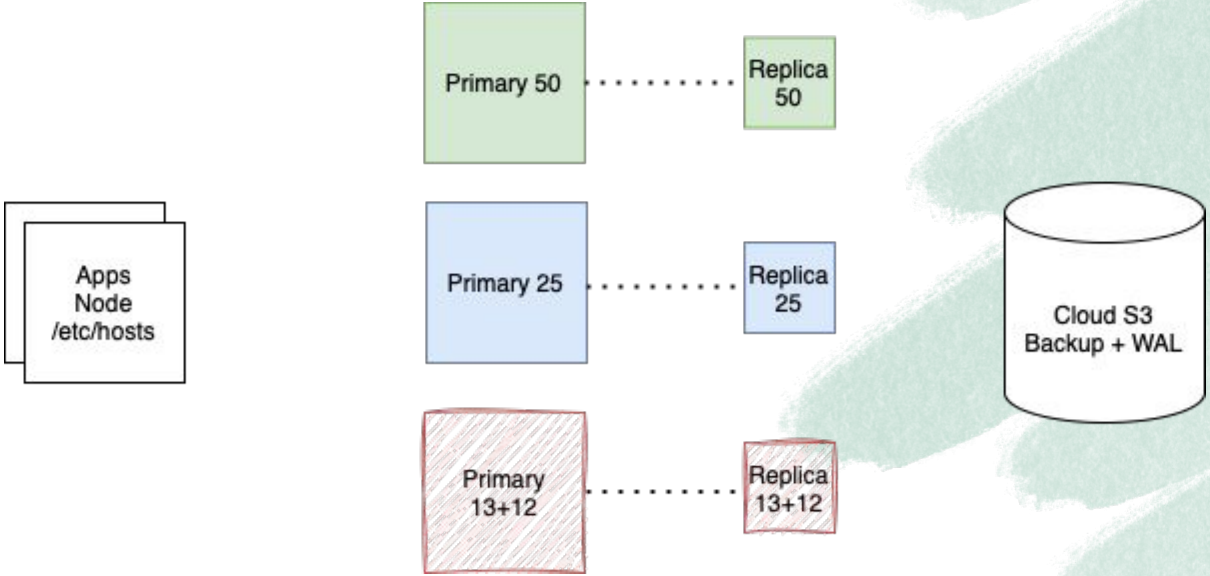
- *max_connections = 400, used around 100*
- We don't need it prior to migration
- Another layer of complexity
- PHP uses permanent connection under the hood (pg_pconnect) + fixed sizes of PHP-FPM pools
- Apps use kind:Service directly in-cluster

Plan: Kubernetes (K8s) Cluster

- Managed Service - take it
- (Managed Control plane-only SaaS also available)

- Standalone cluster for DBs (prefered)
  - +3 VMs for control-plane
  - separated blast radius
  - more management + networking
- Shared with apps
  - at least use .nodeSelector and separate on Node level (noisy neighbours)
  - don't mix apps with DBs on the same node

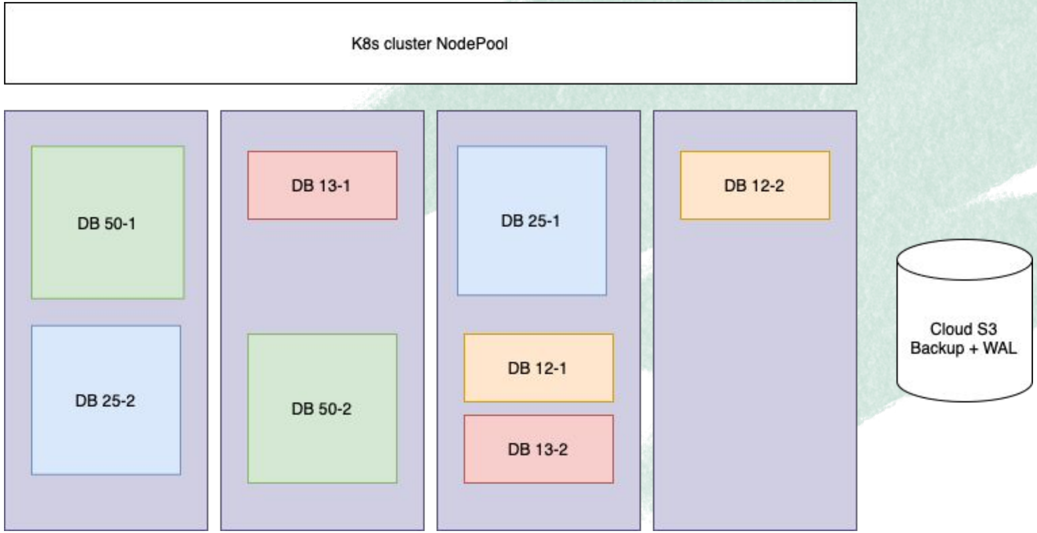
Plan: Node-Pod considerations - VM setup



Plan: Node-Pod considerations

- Our approach: *(Traffic split: 50%, 25%, 13%, 12%)*
- 1 primary+2 replicas? or 1 replica?
- Smaller (single DB Pod) or larger nodes?

Note: we had several incidents on storage infra layer - more replicas won't help.

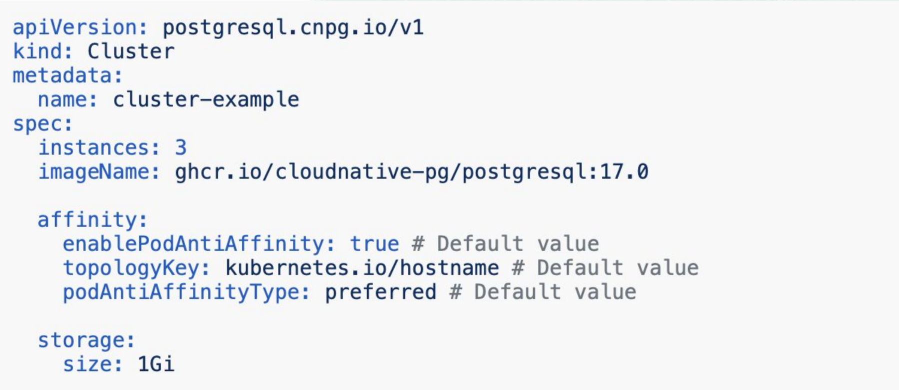


Plan: Node-Pod Affinity

(If possible) schedule Pod to Node that does not contain other Pod like this.

Also considered:

- Any other DB cluster
- cnpg.io/instanceRole: primary



```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0

  affinity:
    enablePodAntiAffinity: true # Default value
    topologyKey: kubernetes.io/hostname # Default value
    podAntiAffinityType: preferred # Default value

  storage:
    size: 1Gi
```

Plan: Node-Pod considerations

- Noisy neighbours conderations
- Bottom line - in an emergency - 2 DBs must share a node
- Considered also separate cluster of "smaller replicas"
- Automatic failover mindset change
- Is it better to use same node and pod sizes, or should we "save $$$"?

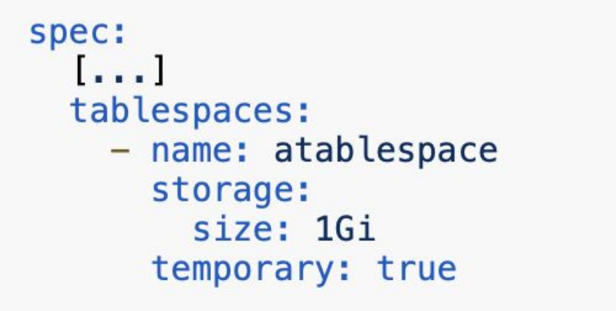
- Great CNPG docs on architectural consideration
- Best-in-class: Shared-nothing architecture

Plan: Disaster Recovery & Backup

- We don't trust OnPrem infra -> barman backup and WAL archive to S3
- <100GB quite easy to download, off-site backup
- DR in cloud from scratch (Terraform managed cluster, GitOps drop-in YAMLs, restore from S3, tested < 40 min) - *client needs several hours for decision*

Note larger DBs or better hosting: CSI snapshotting

Tuning



```
spec:
  [...]
  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
      temporary: true
```

Temp tablespace to a separate partition

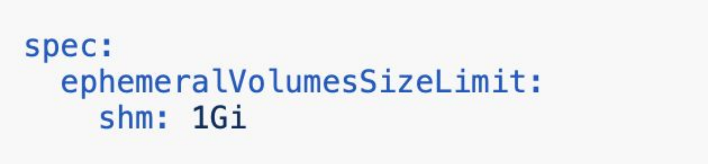
(use local scratch disks)

CPU to HW core allocation (kubelet --cpu-manager-policy)

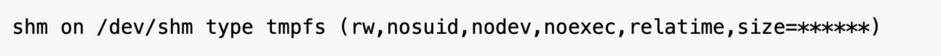
Resource Limits - short story: don't overprovision on PROD

Storage - same logic as for regular VMs

Tuning Postgres



```
spec:
  ephemeralVolumesSizeLimit:
    shm: 1Gi
```



```
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=******)
```

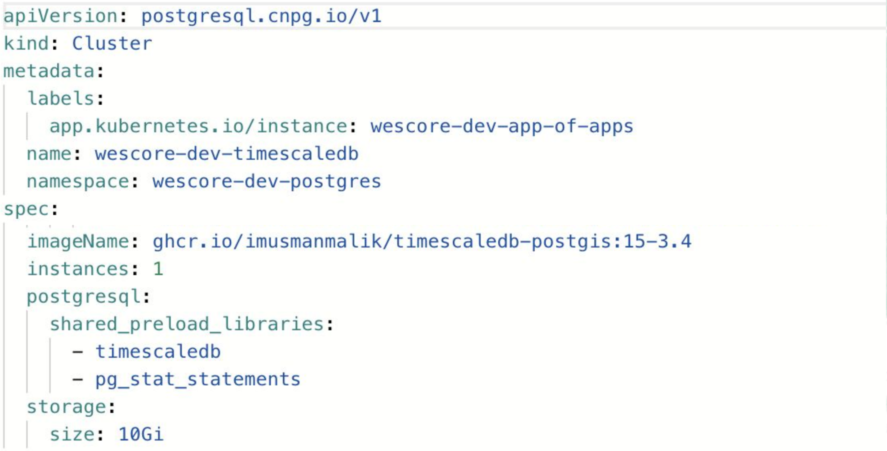
- Shared memory mount
- Direct access to most of GUCs

- Preloaded libs - auto_explain, pg_stat_statements, pgaudit, pg_failover_slots
- pg_repack - requires custom image

- ALTER SYSTEM - limited and discouraged
- Mostly similar to Patroni

Timescale DB, PostGIS, … - possible

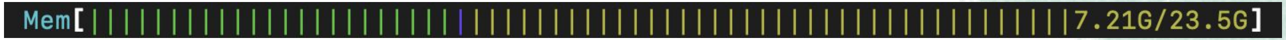
Other extensions also possible via custom PG image



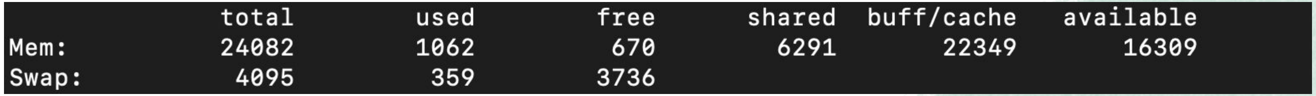
```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  labels:
    app.kubernetes.io/instance: wescore-dev-app-of-apps
  name: wescore-dev-timescaledb
  namespace: wescore-dev-postgres
spec:
  imageName: ghcr.io/imusmanmalik/timescaledb-postgis:15-3.4
  instances: 1
  postgresql:
    shared_preload_libraries:
      - timescaledb
      - pg_stat_statements
  storage:
    size: 10Gi
```

Understanding Pod Memory Usage

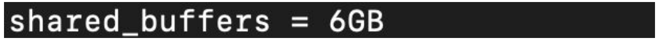
- 2 "schools of Postgres Memory Management"
  - around 25 % of RAM to shared-buffers, let OS handle FS
  - around 80 % of RAM to shared buffers
- VM:



```
Mem[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||7.21G/23.5G]
```

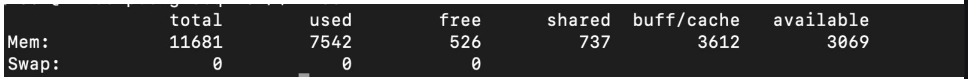


```
              total        used        free      shared  buff/cache   available
Mem:          24082        1062         670        6291       22349       16309
Swap:          4095         359        3736
```

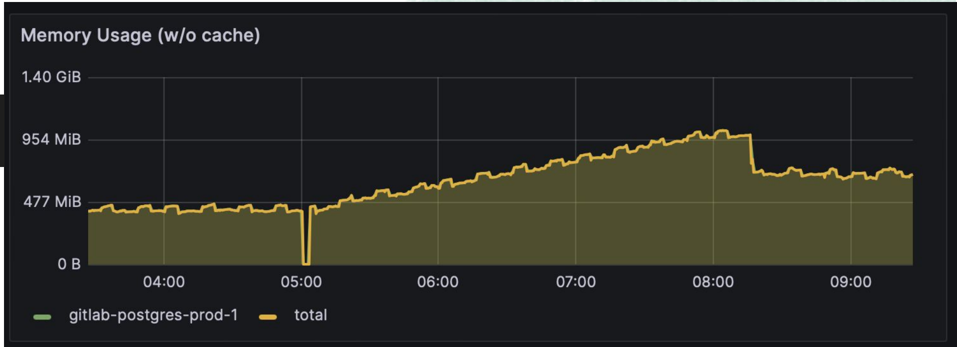


```
shared_buffers = 6GB
```

- Containers:

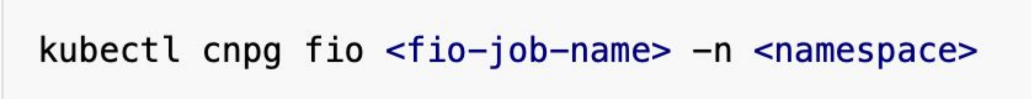


```
              total        used        free      shared  buff/cache   available
Mem:          11681        7542         526         737        3612        3069
Swap:             0           0           0
```

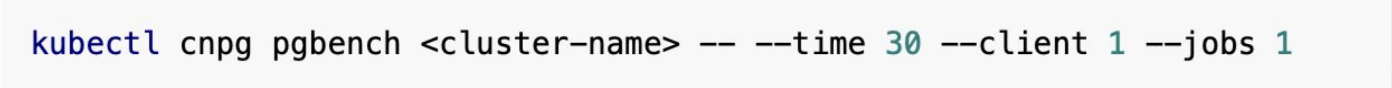


Memory Usage (w/o cache)

Verify: Benchmarking



```
kubectl cnpg fio <fio-job-name> -n <namespace>
```



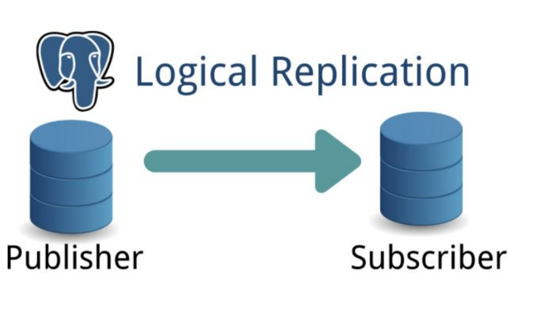
```
kubectl cnpg pgbench <cluster-name> -- --time 30 --client 1 --jobs 1
```

This can't be easier…

Near Zero Downtime Migration

VMs (PG13) -> CNPG (PG16)

- *Create empty cluster*
- *Setup logical replication*
- *Cutover*



G. Bartolini: CloudNativePG Recipe 5 - How to migrate your PostgreSQL database in Kubernetes with ~0 downtime from anywhere

Alternative: Upgrade In-Place and Restore Backup

Upgrade VMs in-place (PG13 -> PG16)

Provision new PROD cluster from backups

Use S3 WAL archive

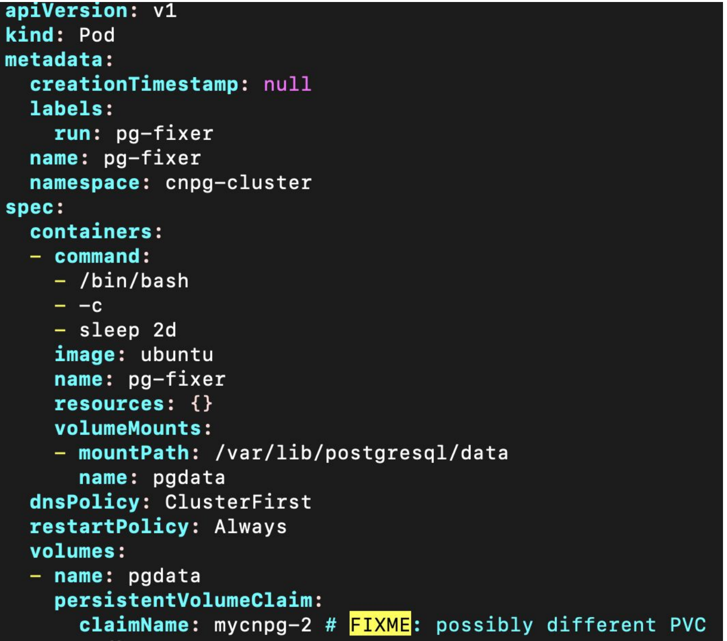
Around 2 hour of downtime

(Same PG version required)

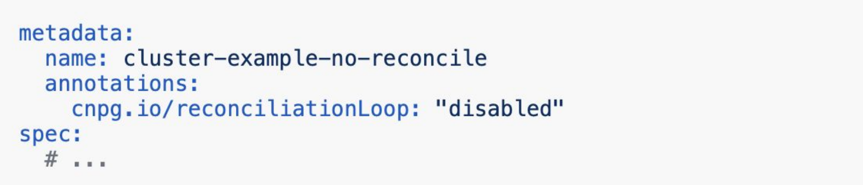
Operator Misbehaving / Break the Glass Scenario

Fencing - marking PG node or cluster - Postgres will remain disabled, Pod runs

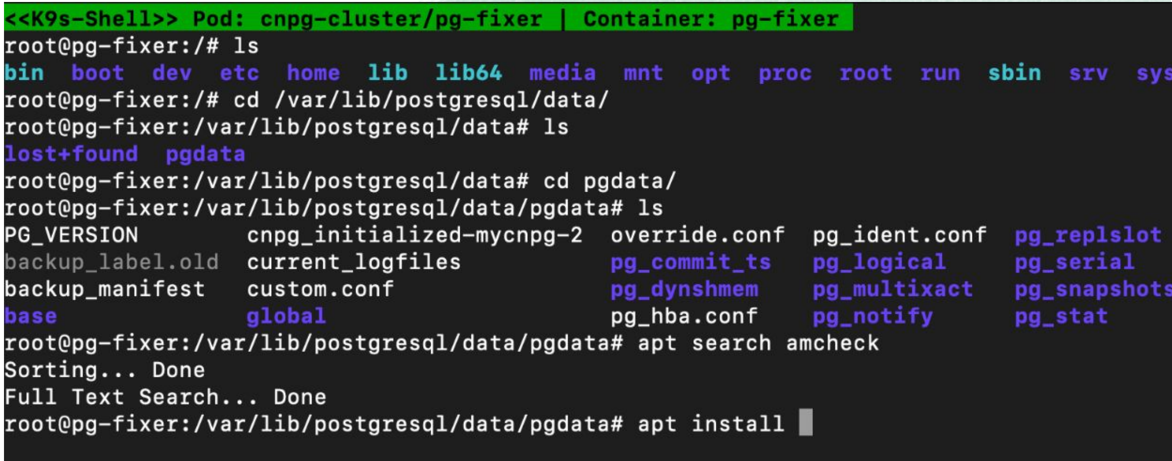
(Not enough for us, Hibernation is too much) - Attach Pod to same PVC - as root



```yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: pg-fixer
  name: pg-fixer
  namespace: cnpg-cluster
spec:
  containers:
  - command:
    - /bin/bash
    - -c
    - sleep 2d
    image: ubuntu
    name: pg-fixer
    resources: {}
    volumeMounts:
    - mountPath: /var/lib/postgresql/data
      name: pgdata
  dnsPolicy: ClusterFirst
  restartPolicy: Always
  volumes:
  - name: pgdata
    persistentVolumeClaim:
      claimName: mycnpg-2 # FIXME: possibly different PVC
```



```yaml
metadata:
  name: cluster-example-no-reconcile
  annotations:
    cnpg.io/reconciliationLoop: "disabled"
spec:
  # ...
```



```
<<K9s-Shell>> Pod: cnpg-cluster/pg-fixer | Container: pg-fixer
root@pg-fixer:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys
root@pg-fixer:/# cd /var/lib/postgresql/data/
root@pg-fixer:/var/lib/postgresql/data# ls
lost+found  pgdata
root@pg-fixer:/var/lib/postgresql/data# cd pgdata/
root@pg-fixer:/var/lib/postgresql/data/pgdata# ls
PG_VERSION        cnpg_initialized-mycnpg-2  override.conf  pg_ident.conf  pg_replslot
backup_label.old  current_logfiles           pg_commit_ts   pg_logical     pg_serial
backup_manifest   custom.conf                pg_dynshmem    pg_multixact   pg_snapshots
base              global                     pg_hba.conf    pg_notify      pg_stat
root@pg-fixer:/var/lib/postgresql/data/pgdata# apt search amcheck
Sorting... Done
Full Text Search... Done
root@pg-fixer:/var/lib/postgresql/data/pgdata# apt install
```

Break the Glass Scenario - Trust the Operator

Let's think about full-autopilot

- SW Bug -> CrashLoopBackOff, verify on UAT
  - Postgres
  - Kubernetes
  - CloudNativePG
- Failover / Switchover, split Brain
  - There are >1 endpoints to kind:Service
  - *(Note edge cases - up to 10s can Pod receive traffic after Endpoint had changed)*
- Reprovision new PG node - around 20 mins on 1Gbit network

Currently still manual:

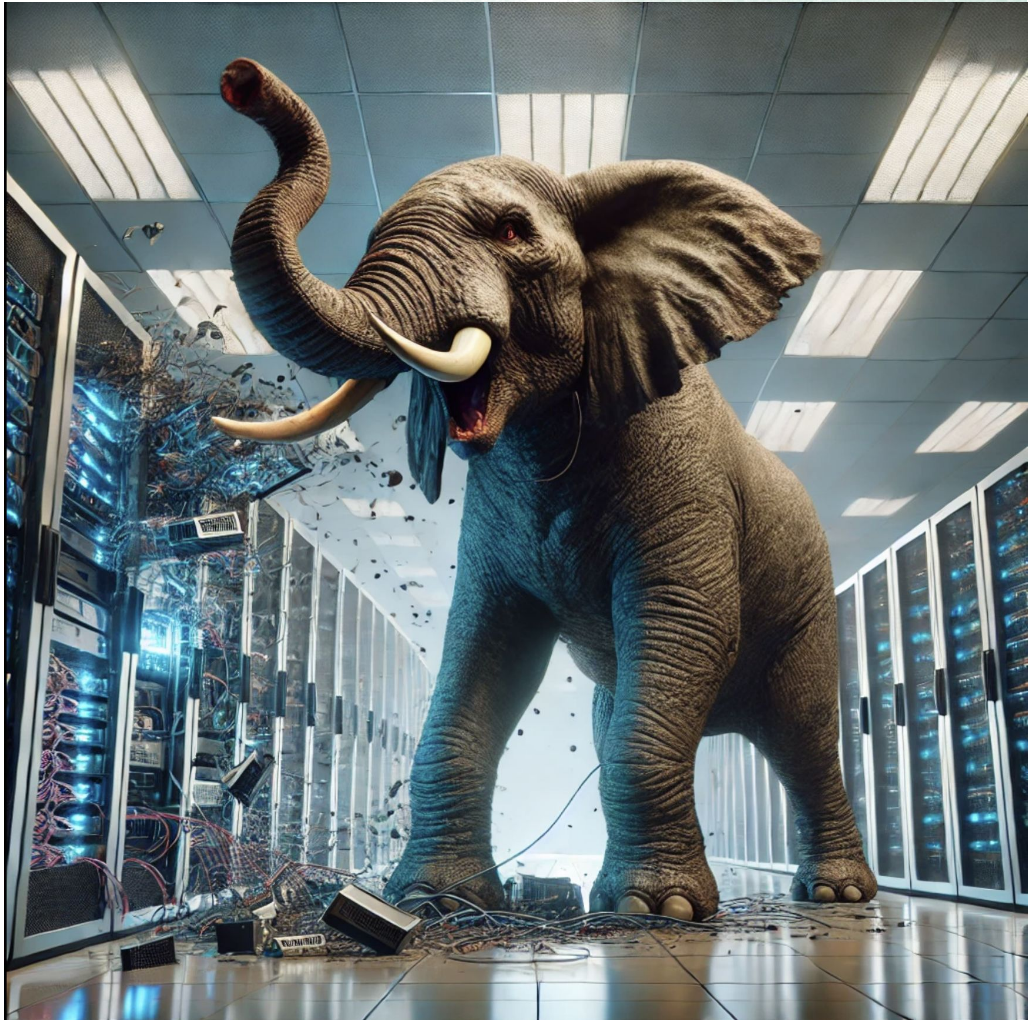
- Password, TLS cert rotation

Horror Stories on PROD

…

nothing here

Just works™



Expected Problems

- Out of disk space -> PVC resize
  - May switch-over depending on the CSI
- Pod restart -> reliability testing
- Node goes down -> reliability testing
- Control Plane goes down (no problems)
- Networking disruptions
- Data corruption -> reliability testing (backups)
- Query Performance problems -> pg_stat_statements

Comparing Before and After

Before:

- Uneven VM sizes (6x)
- 1 manually managed VM per DB
- Ad-hoc managed CPU+RAM

- DR fully manual, never verified
- Backup operations planed in-place
- No updates
- Root access to Devs on VMs

After:

- same Nodes (4x)
- 2-4 DBs per uniform Kubernetes Node
- Large vs Small (½ Large) - 2x increased CPU+RAM for Nodes in total

- Automatic failover
- Backup can be easily bootstrapped next to running PROD, verified and discarded
- Periodic minor version updates
- Pod level access and better insights for Devs

Resume

- K8s nodes easier to maintain to VMs
- Devs basic insight to PG clusters
- GitOps for DB

- Surprisingly easy to use
- Many DBA manual task in YAML instead
  (not time-saving for the first time)

- Several months of research, verification

- Still niche tech (at least in CZ)
- We don't like being early adopters

Next Steps

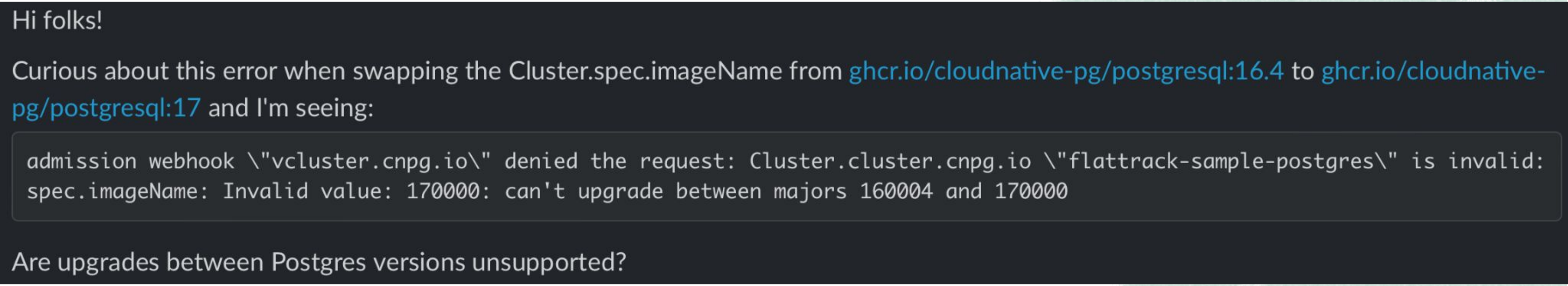
- Offload more traffic on replicas
- Batch data load with locks -> event-driven Kafka
- Performance degradation mitigations

- (With more PROD experience) Offer SLOs to client

- More tooling around Cluster.status (Do we have a fresh backup, …)

It's Still Postgres….

Containers don't change how we should handle it.



Hi folks!

Curious about this error when swapping the Cluster.spec.imageName from ghcr.io/cloudnative-pg/postgresql:16.4 to ghcr.io/cloudnative-pg/postgresql:17 and I'm seeing:

```
admission webhook \"vcluster.cnpg.io\" denied the request: Cluster.cluster.cnpg.io \"flattrack-sample-postgres\" is invalid:
spec.imageName: Invalid value: 170000: can't upgrade between majors 160004 and 170000
```

Are upgrades between Postgres versions unsupported?

We are hiring

Postgres DBA and CloudOps Engineers

Help us manage:
120 product clusters
top DBs 10 kQPS, average prime time load 4-6 kQPS
productuction dataset ~ 27TB (without backups and replicas)

Thank you!